An Overview of the Amoeba Distributed Operating System

Tanenbaum, A.S.; Mullender, S.J.

published in
CWI Syllabus 9: Parallel Computers and Computations
1985

document version
Publisher's PDF, also known as Version of record

AN OVERVIEW OF THE AMOEBA DISTRIBUTED OPERATING SYSTEM

Andrew S. Tanenbaum
Sape J. Mullender

Wiskundig Seminarium
Vrije Universiteit
Amsterdam, The Netherlands

1. INTRODUCTION

As hardware prices continue to drop rapidly, building large computer systems by interconnecting substantial numbers of microcomputers becomes increasingly attractive. Many techniques for interconnecting the hardware, such as Ethernet [Metcalfe and Boggs, 1976], ring nets [Farber and Larson, 1972], packet switching, and shared memory are well understood, but the corresponding software techniques are poorly understood. The design of general purpose distributed operating systems is one of the key research issues for the 1980s.

Several different organizations for distributed computer systems have been proposed [e.g. Newell et al., 1980; Wittie, 1979]. Roughly speaking, these can be classified as personal computer systems, in which each user has a dedicated computer, and systems with a large pool of processors that users can request and return dynamically, as needed. In the former, a six-pass compiler would run sequentially on the user's private machine, whereas in the latter, all six passes could run in parallel on six different machines. The pool-of-processors model is especially attractive because interactive computing is bursty; assigning resources on demand, rather than statically, provides better response for a given investment in equipment. This paper describes the design of a distributed operating system, Amoeba, intended to control a collection of machines based on the pool-of-processors idea. Amoeba has drawn upon the UNIX* operating system [Ritchie and Thompson, 1974] for a certain amount of its inspiration.

2. SERVICES AND PORTS

The basic components of Amoeba are processes, messages, and ports. Processes are active entities, communicating with one another by exchanging messages via their ports. Since simplicity of the operating system is a key goal, the simplest possible interprocess communication mechanism has been chosen: a pure datagram facility. When a process passes a message to the

*UNIX is a Trademark of Bell Laboratories.

operating system for transport, the system makes no guarantee about its delivery, and provides no acknowledgement to the process. Not only does this mechanism greatly reduce the size and complexity of the operating system, but it provides processes with great flexibility in determining their own protocols, flow control schemes, etc. By assuming worst case behavior all the time, unreliable communication media, crashed or migrated processes, and other anomalies become easier to deal with.

The paradigm used in Amoeba for modeling interprocess communication is the **service.** A service is defined by a set of commands and responses, much in the style of an abstract data type. The service is implemented by one or more server processes that accept messages and carry out the requested work. The number of servers per service is determined by the provider of the service, and should not be visible to the users of the service.

Services tend to fall into one of two categories, although the system itself does not make a distinction. On one hand there are public services, such as disk service (reading and writing raw disk blocks), file service (reading and writing files), directory service (file naming and directory management), data base service (relation storage and query processing), time of day, etc. Servers for public services are typically long lived, accepting requests for work, sending replies, and then waiting for the next message. Private services, in contrast, are typically short lived processes started up to run a specific program for a specific user. For example, in the UNIX pipeline a | b | c, b will be started up in such a way as to expect input from a and generate output for c. We will sometimes use the term "process" for private services and "server" for public services, although the system makes no distinction.

Associated with each service are one or more ports used to access the service. When a process, A, wants to communicate with a service, B, A sends a message to one of the ports to which B is listening. To accomplish this, A must know (or be able to find out) the port number If B is a public server, it will usually make its port numbers widely known, but if it is a private service, it will usually keep its port numbers secret.

Knowledge of a port number is taken by the system as prima facie evidence that the sender has a right to use the port. A service may listen to (i.e. accept messages from) any of its ports, and may send messages to any port whose number it knows. All protection in Amoeba is based on port numbers. A little thought will reveal that this mechanism is essentially the same as protection in traditional capability systems [Dennis and van Horn, 1966; Wulf et al., 1974.]. Possession (i.e., knowledge) of a port number allows a process to communicate with the corresponding service, just as possession of a capability allows a process to perform certain operations on the corresponding object.

As an aside, it is interesting to note the fundamentally different paradigm used in UNIX (files), capability systems such as Hydra [Wulf et al., 1974] or StarOs [Jones et al., 1979] (objects), and Amoeba (services). In UNIX all communication is thought of as reading and writing files. In

traditional capability systems, the basic notion is performing operations on objects. In Amoeba, it is sending and receiving messages from protected ports. We believe the semantics of message passing in an unreliable distributed environment to be more natural than those of files or objects, especially with regard to handling lost messages, machine crashes, and flow control. Of course, a service may implement any class of objects it wishes to, so the Amoeba model is similar to the object model.

Since the entire protection system is based on knowledge of port names, it is clear that a method must be found to prevent users from constructing port numbers to which they have no legitimate access. There are two ways to provide such protection. The traditional method is to have the system maintain the port (capability) information on behalf of the users in the form of capability lists. Since user processes do not manipulate the capability lists themselves, there is no danger of them forging capabilities.

The other way is to choose port numbers from a sparse address space. If port numbers are N bits, but only a tiny fraction of the 2^N possible ports are actually used, the chance of a user being able to forge one is small. By chosing N sufficiently large, the probability of trouble can be made arbitrarily small. Of course a very lucky, malicious user might guess an important port number on the first try and bring down the system, but in a traditional operating system the same lucky, malicious user would probably guess the head system programmer's login password, and do just as much damage. In both cases the protection lies in the sparseness of the space to be searched.

At first glance it might seem that encryption could be used to protect capabilities in distributed systems. In fact, encryption is neither necessary nor sufficient. If the capability space is dense, i.e., every possible bit pattern is a valid capability for some operation on some object, a malicious user could simply begin fabricating capabilities at random, with a reasonable chance that many of them could be used to modify or destroy objects scattered around the system. Of course, he would have no way of knowing how well he was doing at wreaking havoc. On the other hand, if capabilities are sparse and encrypted, it will be difficult for anyone to fabricate one, but the protection comes from the sparseness, not the encryption. However, if a few bits in each capability are used to indicate access rights, then encryption will be needed to prevent users from changing the rights. We will discuss this point in detail later.

To keep the amount of protected code to a minimum, we have chosen to make ports sparse, and let user processes manipulate them directly. The format of a typical port is shown in Fig. 1.

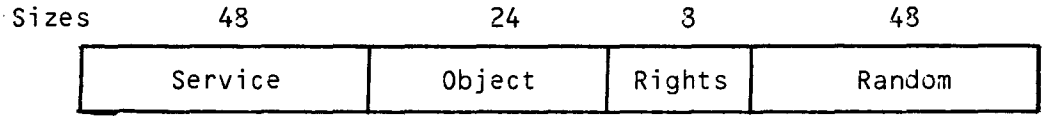

Fig. 1. A Typical port.

The Service field is chosen by the owner of the port. It identifies the specitic service being offered, distinguishing between the many services that may be offered. If a user of the system decides to offer a new service to the public, he will generate a 48 bit random number, and publish that as the name of the service. The same applies for private services, although there the service number will probably not be published. The use of a good 48 bit random number generator, and the fact that only a very tiny fraction of the 2^{48} service numbers will be used makes it very unlikely that two users will accidently choose the same service numbers.

There must of course be different capabilities for receiving messages intended for a service, and sending messages to that service. Several techniques for implementing this exist, one of the simplest being to divide the user population of the system into groups, each group having a unique identification, and to allow reception of messages only on ports with the owners identification in the leftmost 16 bits of the service field.

Each server may allocate the low-order 80 bits as it sees fit. For public services, a common allocation will be 24 bits for an object number, 8 bits for access rights, and a 48-bit random number. A file server, for example, might use the 24-bit object number to identify the file being accessed (in UNIX terms, the disk and inode number). The Rights field can be used to indicate which access rights (e.g., READ, WRITE, RESTRICT, DESTROY) the holder of the port possessed.

3. OVERVIEW OF THE AMOEBA PROTOCOLS

The Amoeba protocols are structured hierarchically. The following sections provide brief overviews of each layer. The layering is shown in Fig. 2.

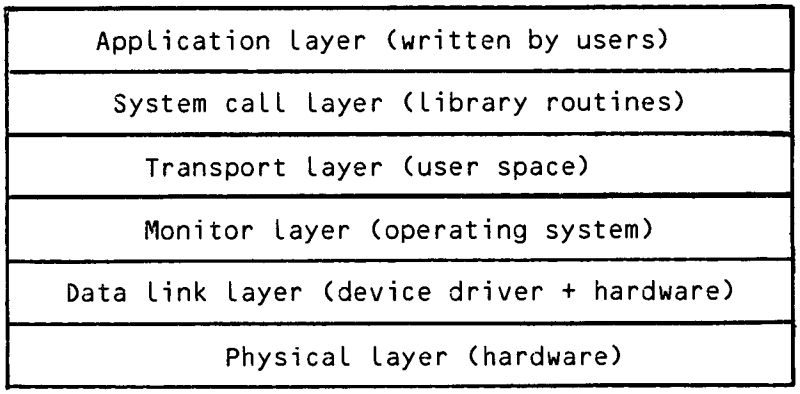

Fig. 2. The Amoeba protocol hierarchy

3.1. The Physical and Data Link Layers

The physical and data link layers are not part of the Amoeba design proper. Ring nets, Ethernets, point-to-point lines using HDLC, etc. are all possible. The only requirement is that they must provide a physical transport facility for moving frames from one machine to another. Error control is handled explicitly by higher layers, so these layers need not provide a reliable transmission facility. Typically these layers would be implemented in a combination of hardware (e.g., Ethernet or HDLC chip) and low level device-driver software.

3.2. The Monitor layer

The monitor layer deals with port addressing. It receives commands from the next highest layer, the transport layer, to send messages to specific ports. It is up to the monitor layer to map the ports onto machine addresses wrap up each message in a packet, and send each packet as a separate entity (datagram). Once a packet has been sent, the monitor layer forgets that it ever existed. The monitor layer has no concept of a connection or any other relation between datagrams. Each one is independent of all the other ones, past and future. Thus our monitor layer is similar to the pups in the Xerox PARC architecture [Boggs et al., 1980] and the internet protocol in the ARPANET [Postel, 1980] except that addressing is done using logical addresses (ports), not physical addresses (machine numbers). To illustrate the difference, if a process migrates, it takes its logical addresses (ports) with it, but not its physical address.

In addition to its role in packet communication, the monitor layer is also involved in low-level process management. Each monitor has a small set of ports used for initiating and migrating processes. A process can only be initiated by sending a process descriptor to a monitor willing and able to run the process. Process migration is a special case of downloading a process initially, with the "core image" coming from another monitor, instead of a disk. If a machine is multiprogrammed, the process and memory mangement associated with the multiprogramming are also part of the monitor layer.

The physical, data link, and monitor layers are part of the operating system kernel, and cannot be modified by the user. If special kernel mode/user mode hardware is available, the monitor should run in kernel mode, whereas the transport layer and higher software would normally run in user mode. If such hardware is not available, other techniques can be used; these will be discussed in a subsequent paper.

3.3. The Transport Layer

The function of the transport layer is to offer a simple and reliable transport service to the higher layers. In order to allow any process to communicate reliably with any other process, its protocol, the transport protocol, should be a system wide convention. Consequently, the code implementing the transport layer, the transport station, could be put into read only memory or EPROM. Technically, any pair of processes are free to use a nonstandard

transport protocol at will, but doing so will normally only be done by people experimenting with transport protocols, not ordinary users.

Without getting into all the details here, the transport layer has been designed with two goals in mind:

1. providing reliable communication to higher layers.
2. making possible a transaction-oriented communication style.

The former is sufficiently well-understood that we will not belabor it here. The intention of the latter is to make it possible for a user to read a file by sending a series of read requests to a file server, but not to require that all requests are processed by the same file server. Such a restriction implies that each request must be self-contained, and that furthermore flow control and error control be handled in such a way as to make switching servers in midstream possible. To achieve these goals, the basic transport service primitives are for users to send a request, and receive a reply; corresponding primitives exist for servers to accept a request, and send a reply.

3.4. The System Call Layer

The System Call Layer provides the user program with a traditional operating system interface. It provides a number of routines that users can call to get the kinds of service provided by most timesharing systems. Typical commands are open file, read file, write file, seek on file, return status of file, close file, create file, remove file, change directory, create process, etc. In most implementations, this layer will be a library package designed to emulate some set of operating system calls. For example, one package might emulate all the UNIX system calls, another package might emulate all the RSX-11 system calls, etc.

These routines carry out their job by sending requests and getting replies from appropriate services in the network. For example, the OPEN system call would typically send a request to a directory server to see if the named file exists and has the correct permissions, with the directory server returning a port through whics hsa ptba orqbl ca iooaffalM Hsa fzfhaj oall layer would normally store the port within itself, for use in carrying out subsequent READ, WRITE, SEEK, STAT, etc. calls on the open file.

3.5. The User Layer

As the name suggests, the user layer is where user programs run. Most user programs will make use of the system call layer to provide a simple and familiar environment in which to run, but since the transport station is within the user's address space, user programs can also use datagrams directly if they choose.

In addition to application programs, a substantial amount of the operating system runs in the user layer. In particular, all of the directory and file system operations, and most of the disk handling, terminal handling,

process management and even accounting are also in the user layer. It is definitely possible for users who have special requirements to implement their own file systems down to the disk block level. Although such private file systems may be incompatible with the standard one(s), they may all coexist on the same disks. How this is accomplished is described in detail later. As a very rough analogy, note that in UNIX, each user may have his own private shell without the system complaining or even caring.

4. THE BASIC FILE SYSTEM

We will now describe in detail one possible way of organizing a file system for Amoeba. As mentioned above, this is by no means the only way. Multiple, incompatible, file systems may coexist peacefully on a single disk during normal operation.

4.1. The Directory Server

The Basic Directory Server (BDS) provides its users with a method for mapping ASCII names onto ports. Each user of the BDS has a home directory which may contain both named ports and named subdirectories. Subdirectories may contain ports for other directories, leading to a general naming graph. Naming conventions are modeled on those of the UNIX operating system, e.g., the path dir1/dir2/file relative to the current directory means the current directory contains a subdirectory "dir1," which in turn contains a subdirectory "dir2," which contains the name "file." Although the BDS may actually maintain a root directory, this directory is not visible to the users, as can be seen from the commands available (below). Put in other words, all paths are relative; absolute paths are not permitted.

BDS has no interest in the meaning of ports, except for directory ports. It simply provides a mapping function from path names to ports. A single directory may hold ports for a mixture of files "owned" by different (possibly incompatible) file servers as well as other types of objects. The ports in a single directory may correspond to files or other objects scattered around a number of machines, in contrast with naming schemes of the form /MachineName/path. The only services provided by the BDS are port storage and UNIX-style naming.

The BDS maintains each directory as a separate file. To do so, it must deal with a file server, of course, but the user need not be aware of these details. The format of a directory is shown in Fig. 3. Note the analogy with UNIX directories, where the port is analogous to an inode number in UNIX.

When a user logs into Amoeba, his shell (command processor) is given the port for the user's home directory. The user may obtain any port reachable from this directory. Privacy is enforced by the lack of absolute path names, i.e., the only ports the user can obtain are those he has made himself and put in one of his subdirectories, or those explicitly given to him by another user. Since one user can pass a port for a directory to another user, multiple links to a single directory may exist, making sharing of a collection of

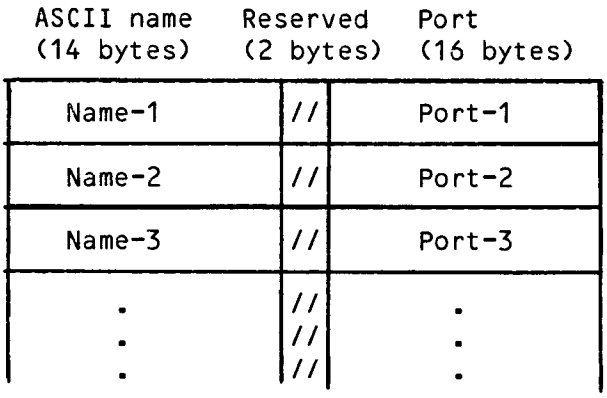

Fig. 3. The Basic Directory Server's directory format.

objects simple. A user may also publish the port for a directory, making its ports known to the general public. Note that a port may only allow a restricted set of operations on its object (e.g., read but not write), so different users may have different access rights to the same object. Which operations are allowed on which ports is determined by the server owning the ports, not by the directory system.

The commands provided by the BDS are listed below. Note that the parameters given correspond to the semantics as seen from the server. In most cases the DirPort parameter will not be explicit, rather it will be the port to which the command is directed. Seen from the user's side, the DirPort will be specified as the port to which the message is sent. For simplicity, we have here assumed that the user makes direct calls on the transport layer. In practice, this "user" will usually be the system call layer.

1. LOOKUP(DirPort, path): port
2. READALL(DirPort): contents
3. ENTER(DirPort, string, port)
4. REMOVE(DirPort, string)
5. MAKEDIR(DirPort, string):port
6. RESTRICT(DirPort, NewRights):NewPort
7. RETRACT(DirPort, NewRights):NewPort
8. RECOVER(AccountPort):data

where
"DirPort" is a port for the directory used by the command
"path" is a path (e.g., subdir/file) relative to DirPort
"string" is a 1-14 character ASCII string not containing "/"
"port" is another port

LOOKUP looks up the path and returns the port. READALL returns the contents of an entire directory. ENTER and REMOVE create and delete directory entries, respectively. Note that the BDS only manages names, not objects. REMOVE, for example, does not destroy the object itself. It is up to the user to destroy unwanted objects, but object creation and destruction is distinct

from naming. In practice, users will typically remove files and other objects by executing the "rm" program, which handles both destroying the object and removing it from the directory system. MAKEDIR creates a new subdirectory. RESTRICT, RETRACT, and RECOVER are discussed below.

The BDS does not provide for garbage collection since it has no way of knowing whether or not a port for a disconnected directory is stored within some executing process or file. The system does provide for disk accounting, so no user can fill up an unlimited amount of disk space with garbage.

As an aside, it is interesting to note that a UNIX directory server is very similar to the one discussed above. A UNIX directory server might have commands

1. LOOKUP(IdPort, FullPath):port
2. READALL(IdPort, FullPath):contents
3. ENTER(IdPort, FullPath, port)
4. REMOVE(IdPort, FullPath)
5. MAKEDIR(IdPort, FullPath)
6. CHMOD(IdPort, FullPath, mode)

where
IdPort is a port used to identify the user
FullPath is an absolute path from the root directory
port is another port
mode is a UNIX protection mode

This server maintains a rooted tree the same as UNIX, with protection based on user and group ids (uid, gid). From "IdPort" the server can determine the user's uid and gid and hence tell whether a requested access is allowed.

4.2. The Flat File Server

Once a user has obtained the port for a file (or other object) from a directory server, it can use the port to perform operations on the file. The directory server is no longer needed, and in fact, it does not matter which directory server provided the port.

As an example of a file server, we consider one whose concept of a file is a linear sequence of bytes, numbered from 0 to the length of the file - 1, with operations to read or write arbitrary bytes, as in UNIX. Other models are certainly possible, for example, a file is a tree of variable length records, with operations to insert and delete records at arbitrary places in the tree. Be sure to note that any user not liking the basic (flat) file system is quite free to provide his own, since the file server, like the directory server, is an ordinary user process.

Associated with each file is a small amount of extra data, not part of the file itself. Directory servers, for example, may use this information to store information needed to recover from lost or garbled directories. Special commands are provided to access the extra information.

The Flat File Server's primitive operations are as follows:

1. READ(FilePort, offset, bytes):data
2. WRITE(FilePort, offset, bytes)
3. READEXTRADATA(FilePort):data
4. WRITEEXTRADATA(FilePort, data)
5. DESTROY(FilePort)
6. STATUS(FilePort):data
7. LOCK(FilePort, key, mode):SuccessFlag
8. UNLOCK(FilePort, key)
9. RESTRICT(FilePort, NewRights):NewPort
10. RETRACT(FilePort):NewPort
11. CREATE(AccountPort):FilePort
12. RECOVER(AccountPort):data

The first five calls are self explanatory. As above, FilePort is typically not present as a parameter but is the local port on the server's side. The sixth call returns the file length, creation date, owner, and other such information. The LOCK and UNLOCK call provide a simple mutual exclusion mechanism. A user can try to exclude other readers or writers or both. Once locked, a file does not respond to commands from ports other than the one that locked it. Numbers 9 and 10 are related to the protection mechanism, as follows.

The flat file server distinguishes four kinds of rights: read, write, lock, and owner. Each of the above commands requires the presence of one or more rights (e.g., DESTROY requires owner+write). The port layout used by the Flat File Server is that of Fig. 1, with the Service field being the same for all files, but the Object field being unique for each file. The object field is an index into a table containing information about the structure and location of each file. (In UNIX terms, it is the inode number.) The Rights field is a bit map containing the rights described above. Since ports are manipulated directly by user code, a mechanism is needed to prevent users from changing their rights.

This mechanism is encryption. When a file is created, the file server builds a 56-bit number with the rights in the first 8 bits and a known constant (e.g., 0) in the low-order 48 bits. The file server then generates a 48-bit random key, stores it in the inode, and finally encrypts the 56-bit number using the 48-bit key. The resulting (56-bit) number forms the rightmost 56 bits of the port. The rest of the port is not encrypted.

When a message is received by the file server, it uses the Object field to locate an inode, and then decrypts the low-order 56 bits of the port to which the message was addressed using the 48-bit key found in the inode. If the decrypted text contains the known constant in the low-order 48 bits, the port is assumed valid, and the rights bits are trusted. If, however, the user has tampered with the port, the chance that the decryption will work is 2^{-43}. Clearly, any encryption algorithm that mixes the bits thoroughly and is immune to a known-plaintext attack will do.

RESTRICT asks the file server to create and return a new port with a subset of the rights in the original port. Thus the owner of a file can give a colleague read but not write access to it.

RETRACT (which requires the owner right) causes a new random number to be put in the inode The corresponding port is returned. It thereby invalidates all existing ports for the file. In most capability systems it is difficult, if not impossible, for the owner of an object to take back capabilities that have previously given away. In Amoeba it is trivial. RECOVER is discussed below.

4.3. The Simple Disk Server

Physical storage of data is completely separate from the file system. The simple disk server has a command to write a (new) disk block and return a port for it. Using this disk address, the file server can read or rewrite the block later. The simple disk server knows nothing at all about file structure.

Disk servers' major customers will normally be file servers. When a user wants to create and write a new file, he will issue a CREATE command to (for example) the flat file server, which will create a new inode and return to the user a port corresponding to it. When the user subsequently writes some data to the flat file server on this port, the flat file server will accumulate a disk block's worth, and then send it to the appropriate disk server. The disk server will then return to the file server the port of the newly written block, so it can be recorded in the inode. Inodes themselves are also stored in disk blocks, although not necessarily with the disk server storing the blocks. For example, inodes might be kept in highly reliable "stable storage" and data in ordinary storage.

In order to store data with a certain disk server, a user needs an account port. Use of the port is prima facie evidence that the user is entitled to disk storage. The object number in the port allows the disk server to locate an entry containing the disk quota, current usage, etc. Each disk block written contains the owner's port, so it is possible to verify if a read or overwrite request for a particular disk block is permitted. File servers normally would reserve the first few bytes of each disk block for information used to recover from lost inodes, for example, the number of the file to which each block belongs, and its position within the file. When a user does a read, the file server gets the entire block from the disk server, but only returns the "real data" part to the user.

The primitive operations provided by the simple disk server are:

1. ASSIGNBLOCK(AccountPort, data):BlockPort
2. READBLOCK(BlockPort):data
3. WRITEBLOCK(BlockPort, data)
4. FREEBLOCK(AccountPort, BlockList)
5. RECOVER(AccountPort):data

ASSIGNBLOCK acquires a new disk block, writes data onto it, and returns a port for it. READBLOCK finds and sends the requested block to the user. WRITEBLOCK overwrites a previously assigned block. FREEBLOCK releases one or more disk blocks no longer needed. RECOVER finds all the blocks belonging to AccountPort and sends the list of block numbers back to the user. It is a very expensive operation, implying a search of a large part of the disk, and is provided to allow file servers to recover from damaged directories. The file and directory servers have analogous recovery operations.

Accounting for disk blocks, and all other resource usage is done by a bank server. When a new user is added to the system by the computer center management, he is given an account with the bank server containing a certain amount of "money." When the user wants disk blocks, he must "buy" them by instructing the bank server to transfer some money from his account to the disk server's. If disk servers refuse to extend credit, a strict disk quota can be enforced.

Although this mechanism can be used for all resources in the system, a number of economic issues arise. For example, is there a single universal currency? In other words, are disk blocks and phototypesetter pages paid for with the same kind of currency, or does one use dinars for disk blocks and zlotys for phototypesetter pages, with no conversion possible? If only one currency exists, there is a danger that some people may use their phototypesetter "budget" for disk blocks, or vice versa, causing some resources to be oversubscribed. If multiple incompatible currencies exist, introducing new resources may imply introducing new currencies as well, which is complicated and inflexible. Another economic issue is what happens when new users are added to the system. Since they must be given some money in order to buy CPU time, disk blocks, etc., the amount of currency in the world tends to increase in time. Should disk servers periodically increase their rates to compensate for this effect (inflation) and if not, how can they prevent scarce resources from being oversubscribed? Perhaps by auctioning them to the highest bidder? These issues are still under investigation.

5. PROCESS MANAGEMENT

There are two situations in which processes are manipulated: when a process is started off from scratch, and when a process moves from one machine to another. Below we discuss the issues involved with both cases.

5.1. Process Creation

In a variety of situations it is necessary for a process to create a new process. For example, the shell needs to create processes to execute commands. Unlike in most operating systems, in Amoeba process creation is not carried out by a system call. Instead, a process wanting to create a child process builds a data structure called a process descriptor, and sends it to a port belonging to a process server. It is up to the process server to carry out the process creation. The Amoeba architecture leaves the question of how process servers are implemented open. They may be application layer processes, or implemented in the monitors.

A process descriptor consists of the following fields:

1. CPU type(s) and options required by the process
2. The port from which the binary file can be fetched
3. The command string
4. The argument strings
5. The environment strings
6. The umbilical port for sending the exit status to the parent
7. The inherited ports

In general, a process will have access to its ports, so it can pass them explicitly to its children. A common situation in which it is important to set up ports is in the shell, when creating pipelines, such as

```
a <infile | b | c >outfile
```

The normal way the shell is expected to create the pipeline is to build three process descriptors, one each for a, b, and c. First, the shell must locate ports corresponding to the object files for a, b, and c. These ports may have a protocol involving some negotiation, since the machine that eventually runs the processes will have to specify which CPU type it is. It is up to the shell to determine in advance which CPU types are available, and put that information into the process descriptor.

The shell then obtains a port corresponding to infile, and inserts that port into the process descriptor for a. It also creates a private port for a's output, and puts that into b, as well as a private port for b's input, and puts that port into a. Finally, two ports are needed for b | c and one for outfile. When all the process descriptors have been built, they are sent to the process server's port for startup.

Notice that ports are used here like UNIX file descriptors. They can be inherited and passed around the same way as in UNIX. Earlier we saw how ports were stored in directories just like inode numbers. The two concepts of inode number and file descriptor have been merged in Amoeba in a clean and simple way: files (or other objects) are always represented by a port which can service them, both in directories and in programs.

5.2. Process Migration

In Amoeba, processes may migrate, to do dynamic load balancing. For example, a process may start out running on a machine that has floating point microcode rather than floating point hardware, because the machine with the floating point hardware was saturated when the process started. Subsequently, the process may migrate to the more appropriate machine (providing the two machines have the same CPU architecture).

Amoeba has been designed to make process migration very similar to starting up a new process. When a monitor wants to get rid of a process, it creates a port from which the binary image can be fetched, and sends the

process descriptor out into the world in search of a new home. When some machine wants to fetch the program, the monitor simply sends it, deletes the port, and discards the process.

6. REFERENCES

Dennis, J.B., and van Horn, E.C.: "Programming Semantics for Multiprogrammed Computations," CACM, vol. 9, pp. 143-155, March 1966.

Boggs, D.R., Shoch, J.F., Taft, E.A., and Metcalfe, R.M.: "Pup: An Internetwork Architecture," IEEE Trans. Commun., vol. COM-28, pp. 624-631, April 1980.

Farber, D.J., and Larson, K.C.: "The System Architecture of the Distributed Computer System - The Communications System," Symp. on Comp. Networks, Polytechnic of Brooklyn, April 1972.

Jones, A., Chansler, R. J., Durham, I., Schwans, K., and Vegdahl, S.R.: "StarOS: A Multiprocessor Operating System for the Support of Task Forces," Proc. Seventh Symp. Operating Sys. Prin. ACM, pp. 117-127, 1979.

Melcalfe, R.M., and Boggs, D.R.: "Ethernet: Distributed Packet Switching for Local Computer Networks," CACM, vol. 19, pp. 395-404, July 1976.

Newell, A., Fahlman, S.E., Sproull, R.F., and Wactlar, H.D.: "CMU Proposal for Personal Scientific Computing," Compcon, pp. 480-483, Spring 1980.

Postel, J.B.: "Internetwork Protocol Approaches," IEEE Trans. Commun., vol. COM-28, pp. 604-611, April 1980.

Ritchie, D.M., and Thompson, K.: "The UNIX Operating System," CACM, vol. 17, pp. 365-375, July 1974.

Wulf, W.A., Cohen, E., Corwin, W., Jones, A., Levin, R., Pierson, C., and Pollack, F.: "HYDRA: The Kernel of a Multiprocessor Operating System," CACM, vol. 17, pp. 337-345, June 1974.

Wittie, L.D.: "A Distributed Operating System for a Reconfigurable Network, Computer," Proc. First Int. Conf. on Distrib. Comp. Sys, IEEE, pp. 669-677, 1979.